Net systems semantics of Web Services Orchestrations modeled in Orc

Sidney Rosario, Albert Benveniste, Stefan Haar, Claude Jard

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1780

NET SYSTEMS SEMANTICS OF WEB SERVICES ORCHESTRATIONS MODELED IN ORC.

SYDNEY ROSARIO , ALBERT BENVENISTE ,
STEFAN HAAR , CLAUDE JARD

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Net systems semantics of Web Services Orchestrations modeled in ORC. *

Sydney Rosario , Albert Benveniste , Stefan Haar ** , Claude Jard ***

Systèmes communicants
Projet DistribCom

Publication interne n ° 1780 — January 2006 — 27 pages

Abstract: Web Services Orchestrations require a firm mathematical basis for their development. We start from the ORC formalism proposed by J. Misra and co-workers, at Austin University. ORC is small and elegant and captures the essence of Orchestrations. We translate ORC into colored Petri net systems, a generalization of Petri nets allowing to handle recursion—this formalism was recently proposed by Devillers et al. Our approach applies as well to standards such as BPEL.

Key-words: Web Services, Orchestrations, semantics, Petri net systems, ORC, BPEL

(Résumé : tsvp)

* This work is supported by RNRT-SWAN contract from the Agence Nationale de la Recherche.
** IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France. E-mail: surname.name@irisa.fr
*** IRISA/ENS Cachan, Campus de KerLann, Rennes, France

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes
Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Sémantique des orchestrations de services Web en termes de systèmes de réseaux de Petri

Résumé : Ce travail présente une sémantique formelle détaillée pour les orchestrations de services Web. Nous partons du formalisme ORC proposé par le groupe de J. Misra à l'Université d'Austin. ORC a l'avantage d'être élégant et compact et de dégager les grandes caractéristiques du concept d'orchestration. Le domaine sémantique choisi est celui des systèmes de réseaux, une extension des réseaux de Petri récemment proposée par l'équipe de Devillers. La même approche peut être appliquée au standard BPEL, dont la traduction est toutefois significativement plus lourde.

Mots clés : Services Web, orchestrations, sémantique, systèmes de réseaux, ORC, BPEL

Contents

1 Motivation

Web Services (WS) Orchestrations and Choreographies have been the subject of numerous studies and standardisation actions [1, 6]. Developing complex orchestrations requires techniques and tools to formally analyse both the functional behavior of an orchestration as well as its Quality of Service (QoS) characteristics. Unfortunately, there are a number of remaining issues regarding WS orchestrations and their formal modeling.

First, the boundary between *orchestrations* and *choreographies* is not clearly established. It is commonly said that choreographies operate on top of orchestrations, aiming at coordinating them. While this objective is clear from the practical standpoint, it does not provide a formally sound difference. Second, tracking queries along their processing by orchestrations is a subtle issue, referred to as *correlations* in BPEL, where the section devoted to correlations is difficult and not really precise [1]. Third, the mechanisms offered for instantiating service calls and other activities also requires some clean foundation: what kind of instantiation? is recursivity offered?

Foundational studies on Web transactions and orchestrations are found, *e.g.,* in [22, 7], using abstract state machines, process algebras, or variants of the π-calculus. Several semantic studies have been performed for BPEL, *e.g.,* [3, 15]. Studies closest to our are [8, 9, 5, 14]; they provide a translation of BPEL into Petri Nets of workflow type (without loops back) aiming at property verification.

QoS for WS is an important but delicate issue. It faces the closed/open world paradox: orchestrations are specified as stand alone "closed" entities. Still, they operate in an open environment, by sharing resources with other orchestrations, other Web Services, and other computing and communication activities. In this respect, orchestrations are just another client of a networked infrastructure.

Some recent work [10, 17, 21, 20] has been devoted to QoS for WS. This work either considers defining and conceptualizing Service Level Agreements (SLA) or address QoS from an experimental viewpoint. There is no work we know that provides formal modeling of WS orchestration QoS in a way that compares with what has been developed for the functional aspects. As a consequence, issues of SLA composition such as mentioned above are not well understood, except for very crude SLA parameterizations.

In this paper we develop semantic foundations for the functional aspect of WS orchestrations, QoS aspects are discussed in companion paper [16].

We provide a clean semantic basis for orchestrations allowing for unbounded but finite recursion, and therefore providing a clean treatment of dynamic instantiation. To clarify the issue, we have chosen to analyse the Orc formalism proposed by J. Misra et al. [11] to specify orchestrations. The interest of Orc is that it is nicely designed, based on few primitive constructs; it implements the so-called "tree-programming" paradigm, where an initial query can be forwarded in parallel and/or cascade to other sites that will contribute to building the answer; answers to partial sub-queries are then progressively collected and eventually returned to the original caller. This paradigm exactly fits the concept of orchestration, it

is more restricted than the model needed to encompass choreographies, where different WS act as peers. Semantic studies for ORC have been developed by Misra et al. [11, 12]. Our semantics targets *Petri net systems* [2] with colors, allowing for a finite representation of infinite nets resulting form dynamic instantiation.

The advantage of this semantics is that a mild extension of it allows us to capture QoS in a mathematically sound way: the formal correlation mechanism that is provided with token colors allows tracking exceptions, and capturing response time is simply performed by adding one more color.

2 The orchestration model: ORC [11, 12]

2.1 ORC, its syntax and intuitive semantics

2.2 Abstract syntax and intuitive semantics

The abstract syntax of the ORC and its intuitive semantics are shown in Table 1. Details of the ORC language are found in [11, 12].

F	Expression name
S	Site
x	Variable
c	Constant
p	Parameter
$x :\in F(p)$	Evaluates $F(p)$, assigns 1st value received to x
$F(p)$	Expression call (new instance thereof)
$S(p)$	Site call, returns at most 1 value or site identifier
$\mathbf{0}$	zero expression; returns nothing and stops
$\mathbf{1}$	one expression; mirrors the result of its previous computation
$\mathbf{let}(p)$	publishes the value of p
$\mathbf{if}(b)$	tests the status of boolean channel b: if true then passes control, otherwise behaves like $\mathbf{0}$
$f \,\|\, g$	Symmetric and concurrent parallel composition: the returns from f and g are interleaved
$f > x > g$	Sequential composition (x optional): each value returned by f causes a fresh evaluation of g and this value can be passed to g via channel x
f **where** $x :\in g$	Asymmetric parallel composition: the 1st value produced by g is passed to f via channel x

Table 1: The abstract syntax of ORC and its intuitive semantics

ORC expressions specify orchestrations; they return zero, one, or a stream of values. In contrast, site calls return at most one value. Timeouts are special sites that raise time based exceptions. The *time* that is referenced in timeouts is the only local time that is attached to the orchestration; no time attached to distant sites is required; this avoids classical inconsistency problems regarding time, caused by distribution. Also, in the ORC model, the only mode for a site call is by *invocation*, service *push* cannot be captured in ORC. Misra et al. call this restricted programming model "tree programming". This paradigm does not exhibit all difficulties of full fledged distributed programming and is still adequate for the simpler case of WS orchestrations—but not for choreographies where orchestrations interact as equal peers. We now present our toy example that will support the rest of our presentation.

2.3 The `CarOnLine` illustrative example

The example is shown in Table 2. The service described consists in getting the pair

(`BestCarPrice`,`BestCredit`)

from the `CarOnLine` service. `CarOnLine` decomposes into the following sequence of operations: 1/ getting, from `CarPrice`, the `BestCarPrice`; 2/ the latter is passed as a parameter to `CreditRate` service, which returns `BestCredit`; if `CarPrice` returns an exception "`Fault`", then the query is reemitted (recursive call). Service `CarPrice` is a broadcast of the same query to a pool of garages. Each garage of the list may return a price or an exception "`Fault`", emitted on time out—note that the `Rtimer` sits on the orchestration site, so that no reference to global time is made. Observe that exceptions are described as part of the orchestration itself; this is the normal way of dealing with exceptions when specifying WS orchestrations.

```
Client :in let(BestCarPrice,BestCredit)
   where (BestCarPrice,BestCredit) :in CarOnLine

CarOnLine =
   CarPrice >BestCarPrice>
      { { if(BestCarPrice!=Fault) >>
          CreditRate(BestCarPrice) >BestCredit>
          let(BestCarPrice,BestCredit) }
      | { if(BestCarPrice=Fault) >> CarOnLine } }

CarPrice =
   broadcast(GarageList) >values> Min(values)
```

`Min(values)` returns `Fault` if `values=Fault` and otherwise it returns the minimum among the tuple of received (valid) values; `broadcast` is defined as follows:

```
broadcast([])   = let(Fault)
broadcast(g:gs) = mux(u,v) where
   u :in {g | Rtimer(timeout) >> let(Fault)}
   v :in broadcast(gs)
```

`mux`, the "multiplexer", is a site call used to filter out the "`Fault`" values due to the timeout; for all valid values `x` and `y`, `mux` is defined as:

```
   mux(x,y) = (x,y)
   mux(x,Fault) = x
   mux(Fault,y) = y
   mux(Fault,Fault) = Fault
```

Table 2: The `CarOnLine` ORC program.

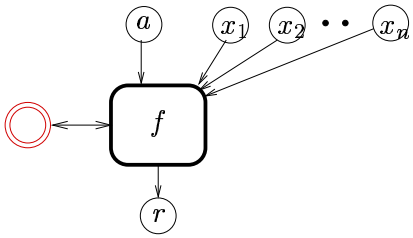
3 Translating ORC into colored Petri nets: principles

In this section, we introduce our overall approach and present its building blocks. Formally, we define the semantics of ORC by a finite representation in terms of systems of (colored) Petri net equations. This is a net-based formalism proposed by [2] that allows describing unbounded nets in a finite manner.

3.1 Reflecting the ORC programming model

The first and most important feature of our translation is that the translation should reflect the ORC programming model. Accordingly:

- Each ORC expression possesses a single activation point. Therefore, its Petri net translation has a special minimal place that we call *activation place*; the start of the execution of an expression corresponds to placing a token in this place.
- Each ORC expression on execution, returns a single (possibly empty) stream of values. Therefore, its Petri net translation has a distinguished place for storing these values, which we call *return place.*
- An ORC expression may use parameters for its execution; The Petri net of an ORC expression possesses one minimal place for each of these parameters; these places are called *parameter places*, they are labeled by their corresponding parameter name.
- The $:\in$ operator in ORC terminates the computation of an expression after it returns its first value. The Petri net translation models termination of expressions by having a distinct place for each expression, called the *power place* of the net. This place keeps track of all the active threads during an execution. An expression may evaluate as long as it has relevant tokens in its power place.
- The translation targets *colored Petri nets* with the following kind of arcs:
  - Ordinary directed arcs, consuming and producing tokens. Directed arcs create causality.
  - *Read arcs*, requiring the presence of tokens in their source node for their sink transition to fire; read arcs do not consume tokens; therefore, they do not create causality and are compatible with concurrency. These arcs are required to model the fact a variable bound to a value may be referenced for an unknown (possibly infinite) number of times.
  - *Reset arcs*, which remove all tokens from their anterior place, disabling the subsequent firings of the posterior transitions.



Generic ORC box

Figure 1: Generic form for the Petri net translation of an ORC expression. The place labelled a is the activation place. The places labelled $x_1, \ldots x_n$ are the parameter places of the ORC expression, i.e the expression is of the form $f(x_1, \ldots, x_n)$. The double-bordered place in red is the power place of the net. The place labelled r is the return place of the net.

Generic form for the Petri net translation of an ORC expression. It is shown in Figure 1. In this figure, the doubly directed arc from the power place to f represents a set of arcs from the power place to transitions in f and from transitions in f to the power place. These arcs could be normal arcs or even reset arcs as we shall see later in the translation.

The arc from the power place to the transition following the activation place denotes the fact that in order to begin execution of an expression, in addition to having a token in the activation place, it is necessary to have a corresponding token in the power place. The tokens in the activation place (and the power place) which activate expressions are called *control tokens* and the tokens in the parameter places which hold the values of variables are called *data tokens*. The power place essentially stores all the control tokens involved in the execution which is later useful in modelling termination. The power place, activation place, transition places and the return place together are called the *interface places* of the net.

Capturing the ORC activation mechanism. The class of nets we use for the translation are essentially colored Petri nets, equipped with a certain *marking equivalence* relation. This design choice is inspired by the work in [2], which aims to build finite Petri nets for recursive expressions. This is essentially achieved by not aiming to build a single net for the whole ORC program but by representing the system as a *collection of nets which may activate each other*. This is often referred to as "net equations" or "net systems" [2].

Each expression E (and so even the sub-expressions of E) has a unique net corresponding to it in this system of nets. The net for an expression E will be denoted by N_E from now on. Note that there may be multiple occurences of E in an ORC program - an expression defined may be called more than once in different parts of the program. Each of these occurences of E is called an *instance* of E. Each instance of an expression will be replaced by a high-level box of the form in Figure 1 in the translation. Furthermore, each instance will have an unique label associated with it, called its *instance label*.

3.2 The Coloring mechanism

Colors. Colors for tokens in our translation serve many purposes: they are used to distinguish different activations of the same expression, to match the control and data tokens in a site call, to model termination of expressions, etc. The color of a token can essentially be seen as a (identifier,value) pair. The identifier component is used in matching related tokens (eg, control and data tokens in a site call) and the value component holds the data carried by the token. We adopt the following conventions for the coloring tokens:

- A single ORC program may be activated more than once in general. Each of these activations are distinguished by a component in the token color, which holds a distinct color for each activation of the main expression of the ORC program. This component would be unnecessary if we consider only a single execution of an ORC expression.
- The parallel composition operator (|) in ORC enables the creation of a stream of values from a single activation. We will need to distinguish the different tokens of a stream and so we append distinct colors to a component in the color of each different control token created in a parallel construct.
- To distinguish the different activations of the same expressions we append the instance label (which is distinct for each instance) of the calling instance to a component of its color while transfering tokens from the activation place (and parameter places) of the instance of E to the activation place (and parameter places) of N_E. The instance label is removed from the token color when the call returns.
- Finally, transitions corresponding to site calls add a color to the token which is the data value returned by the site call.

As a result, we define the color of each token to be a tuple (id,pList,hList,data):

- id is the *identifier, i.e.,* the distinct color added at the start of each different activation;
- **pList** is the list of colors added by the parallel constructs; initially this list is empty, successive parallel constructs append distinct colors to this list;
- **hList** is the list of colors added during expression calls, and
- data is the value carried by the token.

The first three components (*viz* id,pList and hList) correspond to the identifier part of the (identifier,value) pair of the color mentioned previously. Throughout this section,

$$c \quad = \quad (i, p, h, v) = (\mathbf{Id}, \mathbf{pList}, \mathbf{hList}, \mathbf{Data}) \tag{1}$$

shall denote a generic token color as above. For c as above, we shall denote by i_c the i-color of c and so on. For a color c', write simply (i', p', h', v') to denote its components. Finally, if c is indexed, *e.g.,* $c = c_1, c_2, \ldots$, we write $i_1, i_2, \ldots$, for the corresponding i-color.

The matching relation. Colors will play a central role in controlling the firing of transitions. More precisely, to each transition t of net N_E we shall attach a partial function F_t, mapping a tuple of input colors to a tuple of output colors,

$$\text{we call } F_t \text{ the } \textit{firing rule of } T. \tag{2}$$

The domain of each firing rule, *i.e.,* the set of allowed input color configurations, will be expressed in terms of a special family of constraints involving the following *matching relation* defined over pairs of colors:

$$\begin{array}{ll} & c \sqsubseteq c' \\ \Leftrightarrow & (i = i') \wedge (h = h') \wedge (p \in \textit{prefix}(p')) \end{array} \tag{3}$$

where $\textit{prefix}(p')$ is the set of all prefixes of p. The explanation for defining the matching relation in this way is the following:

- $i = i'$: The control and the data tokens should obviously belong to the same initial activation of the ORC program, which is enforced by this condition.
- $h = h'$: This condition ensures that the control and data tokens originate from the same instance. As we know, there may be more than one instance of the same site call and this condition matches control-data tokens corresponding to the same instance.
- $p \in prefix(p')$: Our coloring mechanism ensures that the **pList** components of related control-data tokens satisfy this condition. We observe that the binding of a variable to a value (creation of a data token) can happen either in the sequential composition or in a "where" construct.

  As we shall see, in a sequential composition $f >x> g$, the color of the data token for x is the color of the token returned by f (which is same as the color of the token which activates g). g may produce multiple control tokens corresponding to a single activation (through parallel and/or where constructs) which use this same value of x. The **pList** component of these control tokens will be the **pList** component of the data token for x, possibly appended with one or more colors corresponding to the further branching it underwent in g.

  For the where construct f **where** $x :\in g$ though, the color of the token returned by g (whose value is bound to x), would be different from the color of the token activating the initial expression. Here, when creating the data token for x, we set the first three components of its color to the color of the token activating f **where** $x :\in g$, and its value component is set to the value component of the token returned by g. The exact mechanism is detailed in the translation for the where construct, but note that even in this case, the control tokens in f which use the value of x will have the **pList** of the data token for x as a prefix of their **pList**.

When firing, *transitions must consume or read tokens with matching colors.* These conditions appear as constraints on the transitions and also define the color of the output tokens. Precisely, it is a partial function F_t mapping tuples of input colors to tuples of output colors and is called the *firing rule* of T.

3.3 The marking equivalence

Each ORC program has a *main expression* which is first called when starting an execution (The translation for the main expression is given later in section 4.6). Each activation of an ORC program would involve placing tokens in the activation place of the net for the main expression. Further execution of the ORC program involves calling of the expressions (and sub-expressions) which constitutes it. The modeling of this calling (and return) of expressions is done by using a *marking equivalence* relation between an instance of an expression E which calls it and its corresponding net N_E [2]. This relation is essentially a mapping between the interface places of the instance of E and N_E. The calling of an expression E transfers a token from the activation place of the instance of E to the activation place of N_E.

In general, there may be more than one instance for a given expression E occuring in a ORC program (eg, consider the expression $E \mid E$) while there will be only one net N_E corresponding to all these instances. The unique instance label of each instance will be added to the color of the tokens activating N_E to distinguish tokens corresponding to different activations.

Let I_E denote an instance of E with an instance label l. N_E is the net corresponding to E. a_{N_E} and a_{I_E} denote the activation places of I_E and N_E respectively, PW denotes the power place. Let $c = (i, p, h, v)$ be a token of the form described in section 3.2 and let $c \times P$ denote the marking with token c in the place P and nothing elsewhere. Then for a marking M, the calling of expression E from an instance I_E is given by the following relation :

$$M + c \times a_{I_E} + c \times PW \equiv M + (i, p, h.l, v) \times a_{N_E} + (i, p, h.l, v) \times PW$$

Essentially, the instance label of I_E is added to the **hList** component of the token before transfering it to the activation place of N_E. Strictly speaking, this addition of the instance label to the token color is only needed when there is more than one instance of the same expression, inorder to distinguish the activations corresponding to different instances. If there exists only a single instance of an expression, this addition of the instance label is redundant and can be avoided.

The return of an expression call would move a token in the return place of N_E to the return place of the instance from which it was called. The instance label added during the expression call will be removed from the token color while transfering it back. If r_{N_E} and r_{I_E} denote the return places of N_E and I_E respectively, the equivalence relation for the return of an expression call is given by :

$$M + (i, p, h.l, v) \times r_{N_E} + (i, p, h.l, v) \times PW \equiv M + c \times r_{I_E} + c \times PW$$

where l is the instance label of I_E.

Marking equivalence relations also exist between the parameter places of I_E and N_E. A data token in a parameter place of an instance I_E needs to be transfered to its corresponding parameter place in N_E. For a parameter x, let x_{I_E} denote the parameter place in the instance I_E (with instance label l) and let x_{N_E} denote the parameter place for x in N_E. Then, for a

token $c = (i, p, h, v)$ we have

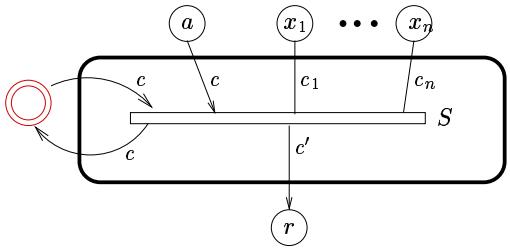
$$M + c \times x_{I_E} \equiv M + (i, p, h.l, v) \times x_{N_E}$$

The only ORC construct that will not contain instances as its sub-expressions are the most basic expression *i.e.,* site calls. As a result, the execution of a site call is completely defined by standard Petri net firing rules, without any marking equivalence relation.

4 The detailed translation

4.1 Site Calls

Sites are the most basic ORC expressions. Site calls may use a list of parameters, *all* of whose values will have to be defined before the site call can happen.



$$\begin{bmatrix} \text{constraint on inputs} \\ \text{output color} \end{bmatrix} = \begin{bmatrix} c_1 \sqsubseteq c \ \ \ldots, \ \ c_n \sqsubseteq c \\ c' = (i, p, h, S(v_1, \ldots, v_n)) \end{bmatrix}$$

Figure 2: Petri net translation of a site call $S(x_1, x_2 \ \ldots \ , x_n)$.

The Petri net corresponding to a generic site call $S(x_1, x_2 \ \ldots \ , x_n)$ is shown in Figure 2. The place labeled a is the activation place of the net. The places labeled $x_1 \ldots x_n$ are the parameter places of the net. The power place is shown in red and return place is labeled r.

Every activation of a site call occurs by the placing of controls tokens in the activation place a. The call will proceed only if the expression which called the site is "active" (i.e has not been terminated) which is ensured by the arc from the power place to the transition following a. For each control token, data tokens with matching colors are needed for the transition of the site call S to fire. As mentioned earlier, this appears as a firing rule for S, detailed in the figure.

For example, the constraint $c_1 \sqsubseteq c$ shown in the figure implies that the token in the place x_1 matches the token in place a.

The value component of the return token is set to the value returned by the site call as shown in the figure (The values of the variables $x_1, ...x_n$ would be the value component in the colors of the tokens corresponding to these parameters, and so the site call $S(x_1, ..., x_n)$

corresponds to $S(v_1, ..., v_n)$). The first three components of the token added to the return place are the same as that of the initial control token activating the site call. A copy of the new token is stored in the power place too as shown.

The arcs from the places corresponding to the parameters, to the transition of the site call are special arcs called *read arcs*. These arcs are required to model the fact a variable bound to a value may be referenced for an unknown (possibly infinite) number of times. Consider the expression $M >x> f \gg S(x)$ where M and S are site calls and f is a recursive expression which returns a stream of infinite values. The value returned returned by M is bound to x which will be used infinitely (each value returned by f causes a new call $S(x)$). Thus consuming a data token during a site call would be incorrect. Also, using normal arcs with a loop instead of read arcs would introduce artificial sequentiality between site calls corresponding to different activations. Read arcs allow us to nicely model the sharing and the non-determinate usage of the data tokens.

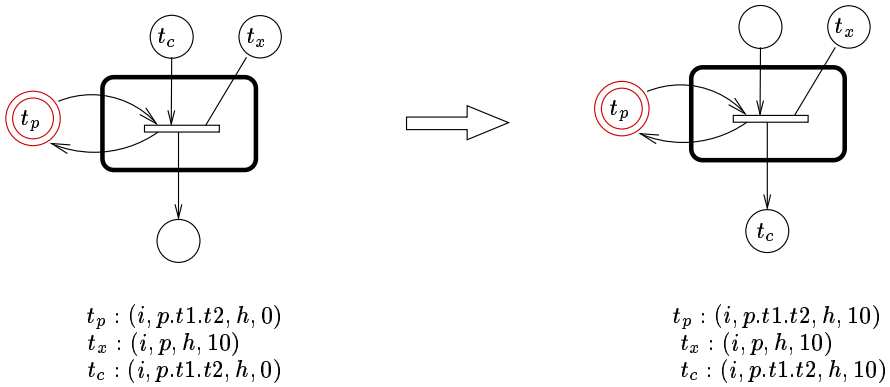


Figure 3: Site Call example : let(x)

An example of a site call $let(x)$ is shown in Figure 3. The arc expressions are omitted for simplicity. The left side shows the system before the site call occurs and the right side is after the firing of the site call transition. On the left, the control token t_c has a token of the same color in the power place (t_p) and the token in the place for the argument x i.e t_x, is $\sqsubseteq$-related to it. On firing of the site call, the token in the argument place remains unchanged (because of the read arc) while the value component of the control token is set to 10 ($let(x)$ returns the value of x which is 10 here) and placed in the return place. The token in the power place t_p is also changed to this new color.

Two special kind of sites defined in ORC are the Constant **0** and **1** sites. The first represents a site that is never called and the second a site that simply mirrors the result of its previous computation. The Petri net translation for these sites is shown in Figure 4. The translations are self-explanatory.

Figure 4: Petri net translation of Constant **0** and **1**.

4.2 Sequential composition

A sequential composition in ORC $f >x> g$ first evaluates the expression f; each value returned by f spawns a new thread of execution for g, with all the occurrences of the variable x in g bound to this value.

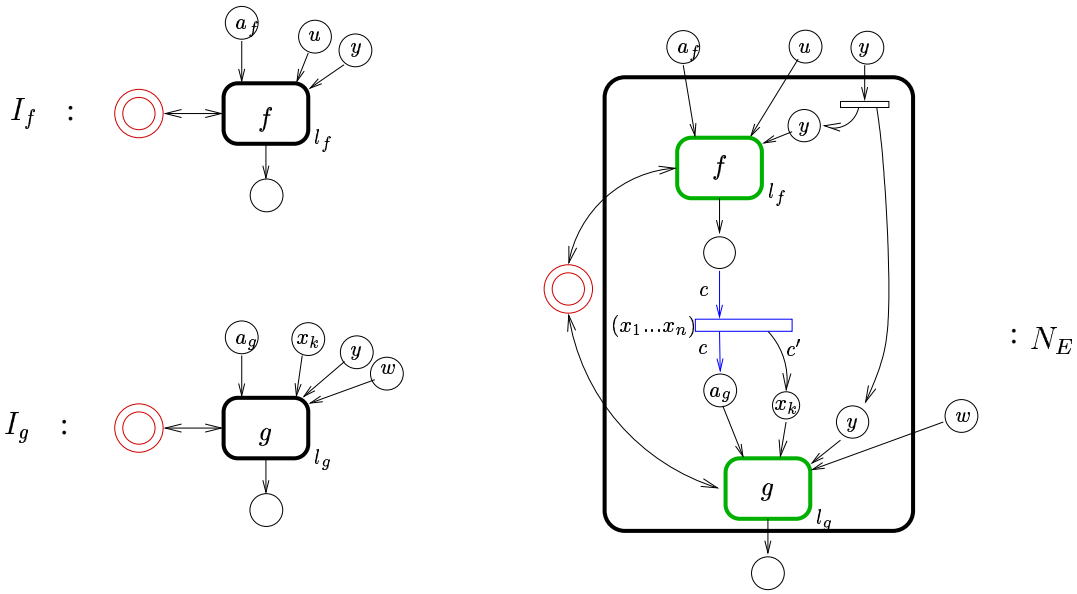


Figure 5: Translation for $f >(x_1, x_2, ...x_n)> g$.

The translation for the generic case of sequential composition $f >(x_1, x_2, ...x_n)> g$ is shown in Figure 5. The left part shows the instances for f and g, I_f and I_g respectively

(having unique labels l_f and l_g, shown on the bottom right of the instance) and the right side shows the net of their sequential composition N_E. I_f and I_g are shown in green in the right side and are linked together with a glue (a transition and two arcs) shown in blue. We assume that the parameter y is common to f and g, the others being distinct. The parameter places for y in I_f and I_g are linked with a single parameter place for y in N_E as shown.

The activation place for N_E is the activation place for I_f and so each activation of activation of N_E will first call f by the marking equivalence relation described previously. When f adds a token to the return place of I_f (again defined by the marking equivalence relation), it is transfered to the activation place of I_g thus resulting in a sequential call to g for every value returned by f. The return place of I_g becomes the return place of N_E.

The value passing is carried out by the blue $(x_1, x_2, ...x_n)$ labeled transition. The value component of the token returned by f (the value returned by f) is to be bound to the tuple $(x_1, x_2, ...x_n)$. The parameter x_k used in g corresponds to the k^{th} component of the value returned by f. Therefore the data component of the token entering the parameter place x_k is set to $comp_k(v)$ where $comp_k(v)$ returns the k^{th} component of a value v.

4.3 Symmetric parallel composition

Symmetric parallel composition of two Orc expressions f and g creates two separate threads for executing them. The output stream of the composition is the time-based merge of the output streams of f and g.

The translation for the symmetric composition is given in Figure 6. The instances for f and g, I_f and I_g are shown on the left hand side. The result of their parallel composition is given on the right hand side, the instances being shown in green here. The activation places of I_f and I_g are linked with the activation place of N_E, allowing simultaneous calling of f and g. The power places are fused and the common parameter places (x in this figure), are linked together as in the case of sequential composition. Finally, the return places are fused so that the stream of values returned by the composed expression is the union of the streams returned by I_f and I_g.

Colors t_1 and t_2 ($t_1 \neq t_2$) are added to the **pList** component of the control tokens entering I_f and I_g, in order to distinguish the different control tokens of a stream when they merge in the return place of N_E. The calls to the nets corresponding to I_f and I_g, the transfer of parameter values and the return of values from them happen according to the marking equivalence relations. Note that since we merge the return places, f and g will add tokens to the same return place, the return place of N_E.

4.4 Asymmetric parallel composition (where expression)

The Orc expression f **where** $x :\in g$ creates two threads which start evaluating f and g in parallel. When g returns its first value, x is bound to it and the computation of g is terminated. f may use the value of x in its execution: portions of f requiring x for its execution will block until it is defined.

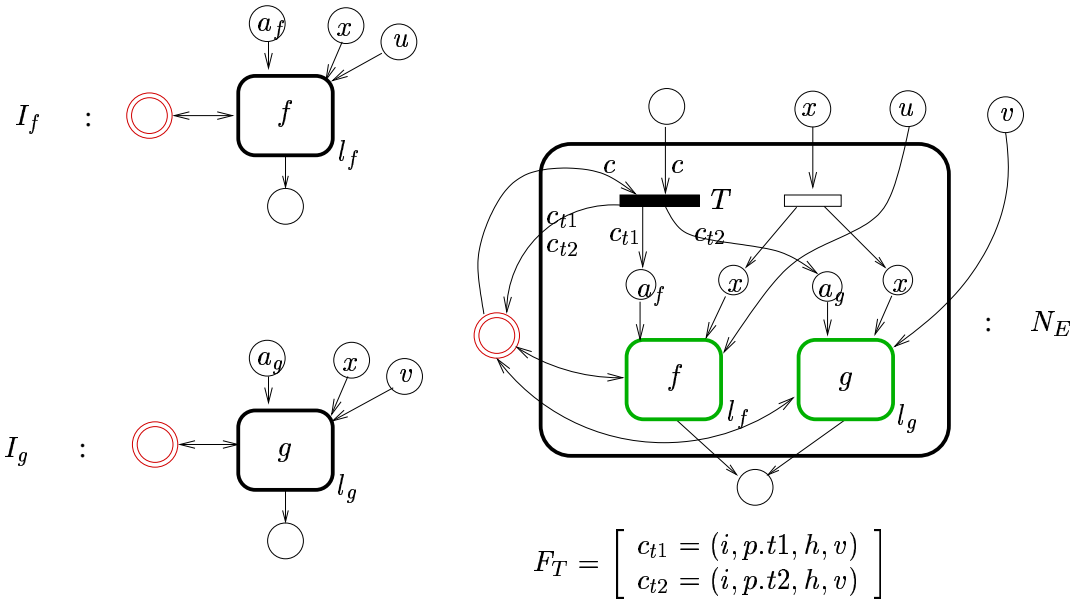
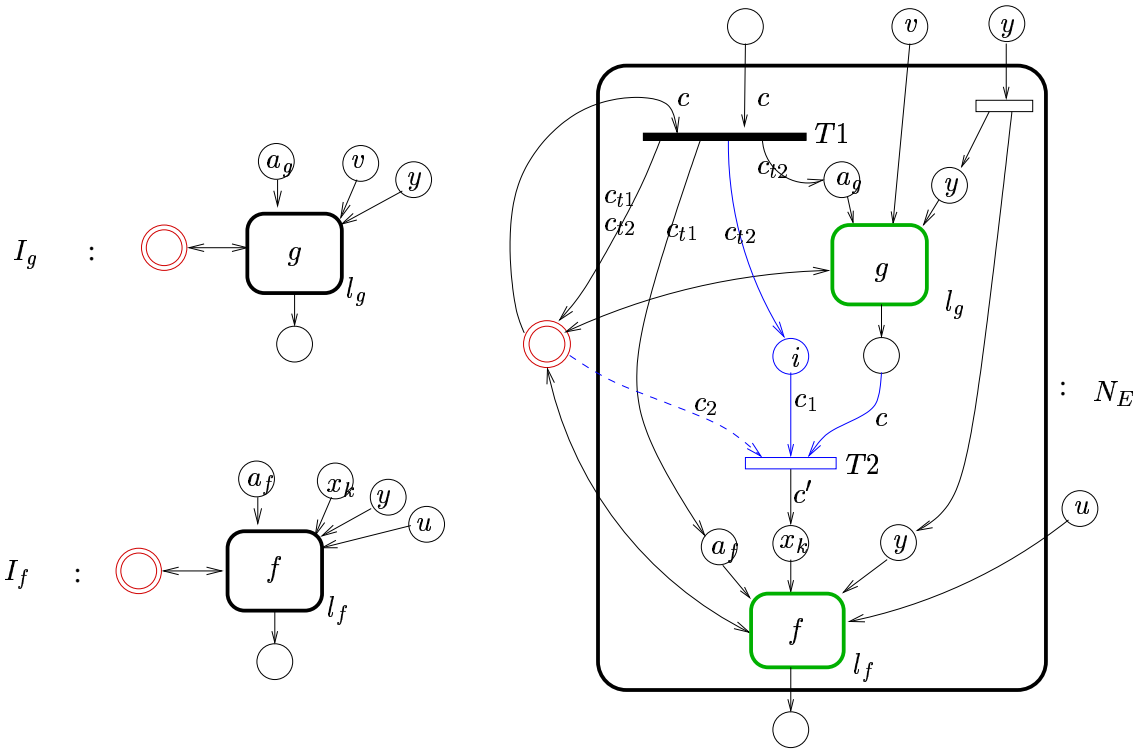


Figure 6: Translating $f \mid g$.

The translation for the generic where expression f **where** $(x_1, x_2, ..., x_n) :\in g$ is shown in Figure 7. The left side shows the instances for f and g, I_f and I_g and the right side shows the net N_E of their **where** composition. The instances of f and g in N_E are shown in green, labeled l_f and l_g respectively. The activation places of I_f and I_g are linked to the activation place of N_E, just like in the parallel composition to enable their simultaneous activation and the control tokens entering them are labelled similarly. The common parameter places (y here) are linked as in the previous constructs and the power places are fused as shown. The addtional net structure in blue helps in selecting the first value returned by I_g and in realising its termination

To understand the termination mechanism we note that when I_g returns a value, there might be other executions corresponding to it occuring at that time. All these executions will have a control token associated with them, a copy of which resides in the power place by means of our construction. Terminating I_g would thus involve removing all these tokens from the power place, disabling the execution of these computations. This is what is achieved by the *reset arc* shown in dashed blue. It removes all the tokens c_2 in the power place which satisfy the condition $c_2 \sqsubseteq' c_1$ where c_1 is the color of the token activating g. The relation $\sqsubseteq'$ is defined as :

$$c_1 \sqsubseteq' c_2 \Leftrightarrow (i_1 = i_2) \wedge (p_2 \in prefix(p_1)) \wedge (h_2 \in prefix(h_1))$$



$$F_{T1} = \left[\begin{array}{l} c_{t1} = (i, p.t1, h, v) \\ c_{t2} = (i, p.t2, h, v) \end{array} \right] \quad F_{T2} = \left[\begin{array}{l} c_1 \sqsubseteq c, \quad c_1 \sqsubseteq' c_2 \\ c' = (i, rem_{t2}(c_1), h, v) \end{array} \right]$$

Figure 7: Translating f **where** $(x_1, x_2, ...x_n) :\in g$.

This differs slightly from the relation $\sqsubseteq$ introduced earlier because here the **hList** components need not be exactly the same; it suffices if the second **hList** is a prefix of the first. This ensures that all the expression calls triggered (directly or indirectly) by the initial activation token c_2 are also terminated.

The function $rem_{t2}(c)$ appearing on the input arc to the parameter place x_k removes the last **pList** component (which is always t_2 for a token in the place i due to our construction) from the token color. As a result, the first three components of the data token for x_k will have the same color as the token c that activated N_E and as a result, the $\sqsubseteq$ relation will hold between the control tokens in f corresponding to the activation token c, and the data token in the parameter place x_k.

The need to distinguish the control tokens arises from the fact that when I_g returns a value and has to be terminated, only the control tokens corresponding to it have to be

removed from the power place. The control tokens for I_f will remain for it to continue evaluating. This can only be achieved by distinguishing their control tokens.

4.5 Expression Definitions

Each occurence of an expression call E is replaced by an instance of it (I_E), a box of the form in Figure 1 with a distinct label to it. The net for that expression N_E is built from the expressions defining E, using the translation rules detailed previously. An expression call is similar to any other call described earlier: when a token appears in the activation place of I_E, we transfer it to the activation place of N_E using our marking equivalence relation. The parameter passing and return of value also occur according our the marking equivalence relation.

In recursive expressions, an instance of the expression call will appear in the net for that expression itself. These expressions are treated as any other normal expression, replacing the expression call by an instance with a distinct label. We give a simple example here. Consider the nets in Figure 8. They correspond to two different states of execution of an ORC program with main expression E where

$$E \triangleq S \gg E$$

S is a site call without any arguments. For simplicity, we have omitted the power place and the arcs from it to the transitions in the net. The labels on the places here denote the tokens that they carry. We have only shown the **hList** component of the token color. Since E is a recursive expression, the net for it N_E contains an instance of a call to E. It is however labeled distinctly ($t2 \neq t1$) and calls from it to N_E can be unambiguously identified.

In the net on the left, the expression call for E is activated in the calling instance by the token with **hList** component $h1$. According to our calling relation for expressions, we append the label of the expression transition instance ($t1$) to $h1$, and move this token to the activation place of the net N_E. As a result the token with color $h1.t1$ is shown in N_E on the right side. A return of an expression call is also shown : the left side has a token $h2.t1.t2$ in the return place of N_E. The label of the last expression call in the **hList** component ($t2$ in $h2.t1.t2$) is removed and the resulting token is placed in the return place of the instance with label $t2$ as shown in the right side (the token $h2.t1$ in the return place of the instance labeled $t2$). Note that actually it is not possible to have a token in the return place of E for this particular example, the reason being that N_E has only one thread of execution with a recursive call in it.

Our translation excludes recursive expressions where an activation of the net N_E causes another activation of it directly by a sequence of marking equivalences. Here the control returns to the activation place *instantaneously* (for, *e.g.,* in the expression, $E \triangleq f \mid E$). This is equivalent to not having any site calls between the activation of a recursive call and the next recursive call caused by that activation. Such kinds of expressions could spawn infinitely many threads "simultaneously", which is not representable using our coding rules. Also, from a practical point of view, such expressions would be unimplementable.

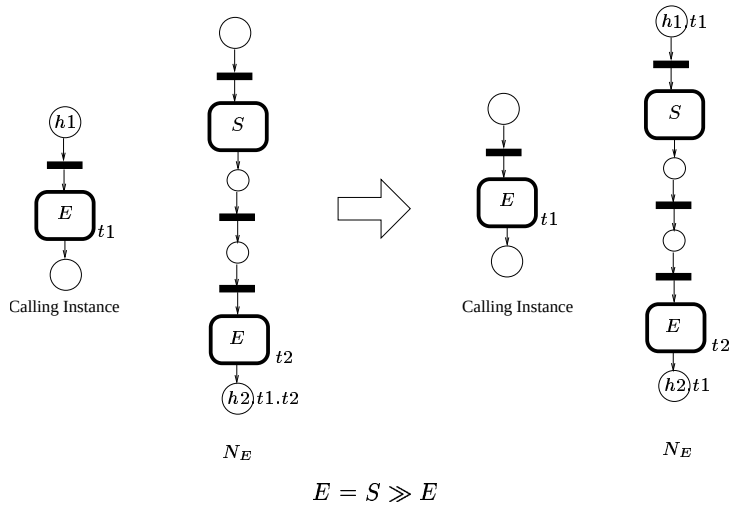


Figure 8: Example showing expression call/return

4.6 The Main Expression

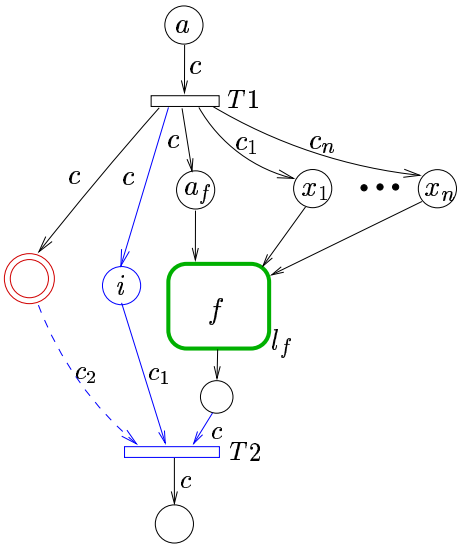
Each Orc program consists of one top level expression—the main expression—which is the starting point of execution of an Orc expression. The presence of this main expression can be likened to the presence of a main program in programming languages (Expression definitions are the equivalent of functions). This top-level statement of an Orc program is of the form

$$x :\in f$$

and essentially does three things : the evaluation of f, the assigning of the first value returned by f (if any), to x and finally the termination of the computation of f when this first value is received (a main expression which does not return a value will never be terminated).

The translation for the main expression is shown in Figure 9. Each new activation adds a token with a distinct color to the activation place labeled a. The outgoing arcs from the transition following it add a copy of this control token to the activation place of the instance of f, a_f, and to the power place to enable execution of f. They also defines the parameters values which will be used by f.

The termination mechanism is achieved by the net shown in blue which is the same as the termination detailed in the **where** expression. It selects the first value returned for each new activation of the main expression and terminates its execution at the same time.



$$F_{T1} = \begin{bmatrix} c_1 = (i, p, h, x_1) \\ \vdots \\ c_n = (i, p, h, x_n) \end{bmatrix} \quad F_{T2} = \begin{bmatrix} c_1 \sqsubseteq c, & c_1 \sqsubseteq' c_2 \end{bmatrix}$$

Figure 9: Net for the main expression

4.7 Net Simplification

The construction detailed here might result in nets which might not satisfy some of the laws obeyed by Orc expressions. For *e.g.*, due to commutativity of $|$, we would expect the nets of $(f \mid g) \mid h$ and $f \mid (g \mid h)$ to be the same (isomorphic?) but this is not attained by this translation. We thus propose to simplify the nets obtained in the translation so that equivalent Orc expressions give the same nets on translation.

In order to facilitate the simplification, we label the transition following the activation place of a net by the kind of the expression it corresponds to. The labels may thus be *site, par* or *where* for nets corresponding to site calls, parallel compositions and where compositions respectively. For the parallel and where compositions, we also label the outgoing arcs from these labeled transitions. The arc to the power place (with the arc expression c_{t1}, c_{t2} in Figure 6 and 7) is labelled p, the arcs to the activation places a_f and a_g in Figure 6 are labelled s and in Figure 7, the arc to a_f is labelled s and the arcs to i and a_g are labelled ns.

The net simplification is described by the following process :

For every transition t labelled *par* or *where* :

Repeat until no further simplification is possible :
If $p \in t^\circ$, connected to t by an arc a labelled s :
If transition $t' \in p^\circ$ is labelled *par* or *where* :
1. For every outgoing arc a' from t' labelled s or ns :
- change originating transition of a' from t' to t
2. remove the arc a from t's outgoing arcs
3. update the arc expression of the outgoing arc of t labelled p to record the new control tokens generated by t

5 Translating the `CarOnLine` example

Figures 10, 11, and 12, show the PN translations of the three components `CarOnLine`, `CarPrice`, and `Broadcast`, respectively. We show the evolution of the colors of a given token when it traverses the different places of the net.

General comments and conventions The following notational conventions are used to describe the firing rule of each transition, cf. (2). Color c_1 decomposes as $c_1 = (i_1, p_1, h_1, v_1)$ and similarly for other color names. A distinct label is attached to each transition, *e.g.*, $T_1, T_2, \ldots$ Transitions denoting site calls are white; other transitions, added for the purpose of the translation, are black.

For each transition, we give in a separate table the firing rule as a set of constraints on input colors (*e.g.*, $v_1 = []$ for T_1 and T_2 in Figure 12) and the equations giving the resulting set of output colors. In these equations, names of colors are local to each transition and are specified in the diagrams; for example, referring to Fig. 12, transition T_3 has c and c_1 as input colors and c' as output color. Since names are local to transitions, they are reused across the diagram without referring necessarily to identical colors.

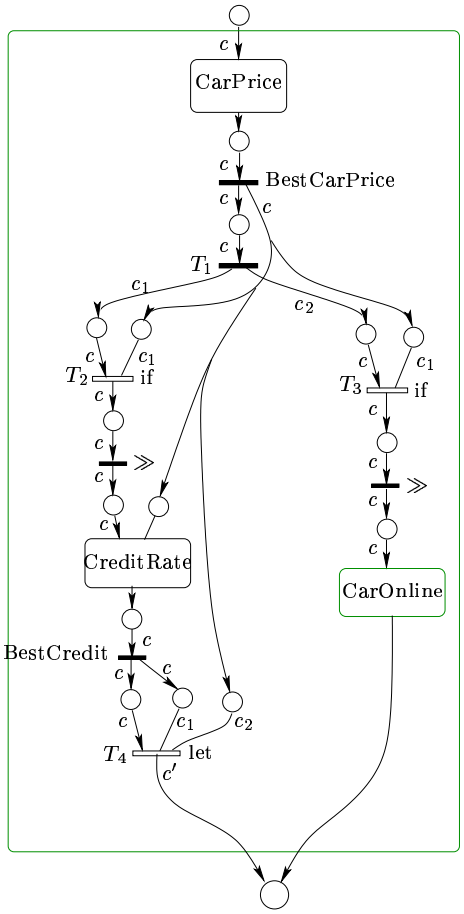
For each transition, the activation token is systematically denoted by c; the constraint $c' \sqsubseteq c$ always holds, where c is the color of activation token and c' is the color of any input token; to simplify the diagrams, these systematic constraints are not mentioned.

The power place and its related arcs are omitted from Figures 10, 11, and 12. Detailed comments follow, for each figure.

Figure 10 A simplification has been performed in this figure. The arc leaving the transition BestCarPrice, which is shown as branching into four arcs, actually represents four different arcs with the same arc label (i, p, h, bcp). The parallel appends a label $t_i, i = 1$ or 2 to the **pList**, for its two branches.

Figure 12 This figure exhibits a "**where**" expression with its two "$:\in$" subexpressions located at transitions T_{10} and T_{11}. These two transitions extract only the first token in case a stream of tokens is generated—this may for example occur when an expression is called

in the scope of the **where**; *e.g.,* here `Broadcast` is re-called. The mechanism for achieving this is by matching token colors: according to (3), two tokens can synchronize at the "$v :\in$" transition iff they possess identical **hList** color, *i.e.,* $h' = []$; this constraint selects only the 1^{st} token produced.



T_1 : $\left[\begin{array}{l} c_1 = (i, p.t_1, h, v) \\ c_2 = (i, p.t_2, h, v) \end{array} \right]$ T_2 : $[\ v_1 \neq \texttt{Fault}\ ]$

T_3 : $[\ v_1 = \texttt{Fault}\ ]$ T_4 : $[\ c' = (i, p, h, (v_1, v_2))\ ]$

Figure 10: `CarOnLine` and its firing rules. Note the recursive call of `CarOnLine`.

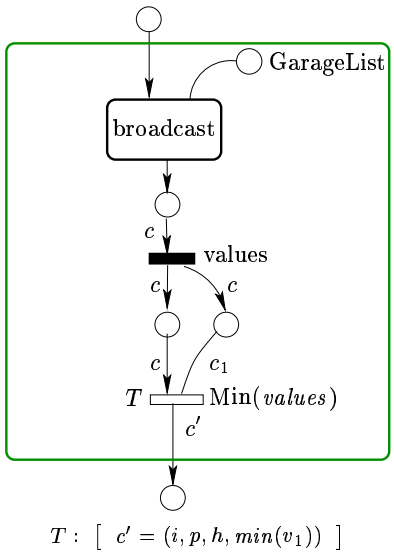
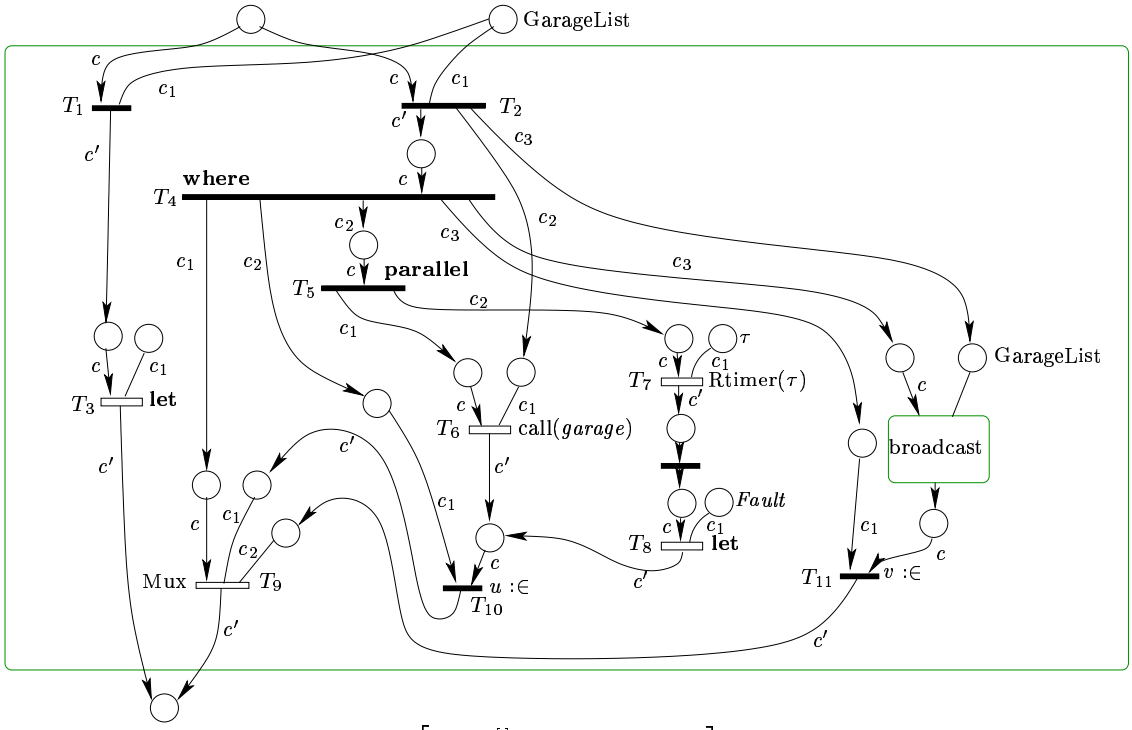


Figure 11: `CarPrice` and its firing rules.



T_1 : $[\ v_1 = [\,]\ ]$

T_2 : $\left[\begin{array}{l} v_1 = [\,] \\ c' = c \\ c_2 = (i_1, p_1, h_1, head(v_1)) \\ c_3 = (i_1, p_1, h_1, tail(v_1)) \end{array}\right]$

T_3 : $[\ c' = (i, p, h, v_1)\ ]$

T_4 : $\left[\begin{array}{l} c_1 = (i, p.t_1, h, v) \\ c_2 = (i, p.t_2, h, v) \\ c_3 = (i, p.t_3, h, v) \end{array}\right]$

T_5 : $\left[\begin{array}{l} c_1 = (i, p.t_4, h, v) \\ c_2 = (i, p.t_5, h, v) \end{array}\right]$

T_6 : $[\ c' = (i, p, h, call(v_1))\ ]$

T_7 : $[\ c' = (i, p, h, v_1)\ ]$

T_8 : $[\ c' = (i, p, h, v_1)\ ]$

T_9 : $[\ c' = (i, p, h, \mathrm{Mux}(v_1, v_2))\ ]$

T_{10} : $\left[\begin{array}{l} c = (i, p.t_2, h, v) \\ c' = (i, p, h, v_1) \end{array}\right]$

T_{11} : $\left[\begin{array}{l} c = (i, p.t_3, h, v) \\ c' = (i, p, h, v_1) \end{array}\right]$

Figure 12: `BroadCast` and its firing rules. Note the recursive call of `Broadcast` with updated parameters.

6 Conclusion and future work

We have formally defined the functional semantics of Web Services Orchestrations. This framework relies on an abstract model in the form of colored Petri net systems and encompasses recursion. To make the presentation of things cleaner, we have chosen to build on top of the Orc formalism; however, the same principles would apply to BPEL.

The advantage of this semantics is that a mild extension of it allows us to capture QoS in a mathematically sound way: the formal correlation mechanism that is provided with token colors allows tracking exceptions, and capturing response time is simply performed by adding one more color. This extension is presented in the companion paper [16].

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte (Editor), I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services. [BPEL4WS.] Version 1.1. 5-May-2003. 136 pages. http://xml.coverpages.org/BPELv11-May052003Final.pdf

[2] R. Devillers and H. Klaudel. Solving Petri Net Recursions Through Finite Representation. Proc of IASTED'04.

[3] J. Arias-Fisteus, L. Sánchez Fernández, and C. Delgado Kloos. Applying model checking to BPEL4WS business collaborations. SAC 2005: 826-830.

[4] I. Grosclaude, C. Dousson, P. Le Maigat, and R. Trinquart. Scénarios et modèles d'orchestration. Swan report SWAN/WSM/LIV/1.

[5] S. Hinz, K. Schmidt, and C. Stahl: Transforming BPEL to Petri Nets. Business Process Management 2005: 220-235.

[6] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon. Web Services Choreography Description Language – WS-CDL, version 1.0. http://www.w3.org/2002/ws/chor/edcopies/cdl/cdl.html

[7] C. Laneve and G. Zavattaro. Foundations of Web Transactions. FoSSaCS 2005: 282-298.

[8] A. Martens. Analysis and re-engineering of Web Services. *Proc. of 6th International Conference on Enterprise Information Systems* (ICEIS'04). Porto, Portugal, 2004.

[9] A. Martens, Ch. Stahl, D. Weinberg, D. Fahland, Th. Heidinger: Business Process Execution Language for Web services - Semantik, Analyse und Visualisierung. Informatik-Berichte Nr.169, Humboldt-Universität zu Berlin, July 2004.

[10] D. A. Menascé, V. A. F. Almeida. *Capacity Planning for Web Services: metrics, models, and methods.* Prentice Hall, 2001.

[11] Jayadev Misra. Computation orchestration: a basis for wide-area computing. A preliminary version of this paper will appear in the Lecture Notes for NATO summer school, held at Marktoberdorf in August 2004. http://www.cs.utexas.edu/users/psp/Wide-area.pdf

[12] J. Misra and W. R. Cook. Orchestration in Orc: A Deterministic Distributed Programming Model. (An extended abstract, unpublished, April 2005). http://www.cs.utexas.edu/users/psp/OpSemantics.Orc.pdf

[13] W. R. Cook and J. Misra. Implementation Outline of Orc. See www.cs.utexas.edu/users/wcook/projects/orc/

[14] C. Ouyang, E. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas, and A.H.M. Hofstede. Formal semantics and analysis of control flow in WS-BPEL. Research report, Queensland university of technology, 2005. http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2005/BPM-05-15.pdf

[15] W. Reisig: Modeling- and Analysis Techniques for Web Services and Business Processes. FMOODS 2005: 243-258.

[16] S. Rosario, A. Benveniste, S. Haar and C. Jard. Foudations for Web Services Orchestrations: functional and QoS aspects, jointly. Submitted for publication. 2006.

[17] A. Sahai, V. Machiraju, M. Sayal, Aad P. A. van Moorsel, and F. Casati. Automated SLA Monitoring for Web Services. DSOM 2002: 28-41.

[18] K. Schmidt and Ch. Stahl. A Petri net semantic for BPEL4WS—validation and application. In E. Kindler (Ed.): *Proc. 11th Workshop on Algorithms and Tools for Petri Nets* (AWPN'04), Paderborn, 2004, pp. 1-6.

[19] K. Schmidt. Distributed Usability of Web Services. In E. Kindler (Ed.): Proc. 11th Workshop on Algorithms and Tools for Petri Nets (AWPN'04), Paderborn, 2004, pp. 19-24.

[20] H. G. Song and K. Lee: sPAC (Web Services Performance Analysis Center): Performance Analysis and Estimation Tool of Web Services. Business Process Management 2005: 109-119.

[21] M. Tian, A. Gramm, T. Naumowicz, H. Ritter, J. Schiller. A Concept for QoS Integration in Web Services. 1st Web Services Quality Workshop (WQW 2003), in conjunction with 4th International Conference on Web Information Systems Engineering (WISE 2003), Rome, Italy, December 2003.

[22] M. Viroli. Towards a Formal Foundation to Orchestration Languages. *Electr. Notes Theor. Comput. Sci.* 105: 51-71 (2004).